DIC Final Project Report

通訊四 407430013 林子翔

June 8, 2022

Intoduction

In this project, we want to design an floating-point number multiplier chip, which complies with the IEEE-754 standard on 64-bit basic double format with "rounding to nearest" mode.

At the first section, I will describe my design on the algorithm, and then I will explain the method to check my verilog program. At the third section, I will show the implementation result and the simulation statistics.

1 Algorithm

Our multiplier circuit can be simply divided into three stages:

1. Input stage
2. Calculation stage
3. Output stage

Since the input stage and output stage just use a counter to move in or move out the data, we will mainly focus on how the calculation works.

To describe my code, some used variables are listed here.

```
reg [63:0] A, B;     // input data
reg subnormal;       // subnormal number indicator
reg sign;
reg signed [12:0] expn;
reg [105:0] mprod;
```

At the end, {`sign, expn[10:0], mprod[103:52]`} would be output.

* RTL Code Location: `/misc/Si2_RAID-1/COURSE/dic/dic03/final_project/fp_mult.v`

1.1 Calculation

To clearly state the progress of the algorithm, I am going to describe the calculation cycle-by-cycle.

Clock cycle 0

At this time, the input data have located in `A` and `B`.
In the first cycle in calculation stage, we have to check whether `A` or `B` represents some special value, such as NaN and infinity.
If we are in one of the following conditions, raise the `calend` flag and assign `sign`, `expn`, `mprod` to corresponding value. As the flag raised, it would enter the output stage at next cycle, instead of going to the **Clock cycle 1** we would mention later.

```
// A is NaN
if (A[62:52] == {11{1'b1}} && A[51:0])
    calend <= 1;
// B is NaN
else if (B[62:52] == {11{1'b1}} && B[51:0])
    calend <= 1;
// A is 0 and B is infty
else if (!A[62:0] && B[62:52] == {11{1'b1}} && !B[51:0])
    calend <= 1;
```

```
// B is 0 and A is infty
else if (!B[62:0] && A[62:52] == {11{1'b1}} && !A[51:0])
    calend <= 1;
// A or B is 0 and both not infty
else if (!A[62:0] || !B[62:0])
    calend <= 1;
// A and B are both subnormal numbers
else if (!A[62:52] && A[51:0] && !B[62:52] && B[51:0])
    calend <= 1;
```

Simultaneously at cycle 0, if `A` represents a subnormal number, `A` and `B` would be swapped. With this trick, the subnormal number would be always fixed at `B`. Afterwards, if we see `subnormal` flag is raised, we would know that `A` is a normal number and `B` is a subnormal number, without checking again.

Clock cycle 1 - 4

If we did not leave calculation stage at **clock cycle 0**, there are two possibilities:

- `A` and `B` are both normal numbers if (`!subnormal`).
- `A` is normal and `B` is subnormal if (`subnormal`).

When it comes to multiply two floating number, there are three basic operations we need to do:

- xor their **sign bit**
- add their **exponential part**
- multiply their **fractional part**

During the four cycles, we want to complete the multiplication part. But since they have 52 or 52+1 bits (+1 for the implicit leading 1 of normal numbers), it is a huge size for binary mutiplication. To prevent overloading one single cycle, we divide this computation to four cycles, as follow:

```
else if (inend && !calend && counter == 4'b0001) begin  // 1
    mprod <= {1'b1, A[51:0]} * B[13:0];
end
else if (inend && !calend && counter == 4'b0011) begin  // 2
    mprod <= mprod + (({1'b1, A[51:0]} * B[26:14]) << 14);
end
else if (inend && !calend && counter == 4'b0010) begin  // 3
    mprod <= mprod + (({1'b1, A[51:0]} * B[39:27]) << 27);
end
else if (inend && !calend && counter == 4'b0110) begin  // 4
    mprod <= mprod + (({1'b1, A[51:0]} *
                          {~subnormal, B[51:40]}) << 40);
end
```

If (`subnormal`), we need to do an additional thing during these four cycles. That is, indexing the highest `1` in the fractional part of `B`. We want a result like this:

```
The position of highest 1 in B[51:0]:
#####################################################
|                                                   |
|                                                   |
1                    - result -                    52
```

The kind of indexing is a little bit compilicated in verilog HDL. To achieve this, I try to do binary search at first and then apply bitwise searching. At the beginning, we need to introduce some additional registers:

```
reg [5:0] idxMsb;
reg [2:0] msb_at_block;
reg [25:0] tmpbuf;  // temp buffer
```

For example, assume that we have such `B[51:0]`:

```
0000000001000_0000000000000_0000000000000_0000000000000
          |
          9
```

At cycle 1, `B[51:0]` is compared with `(1 << 26)`, and `msb_at_block[2]` is set since `B[51:0] >= (1 << 26)`.

```
msb_at_block:
1xx
tmpbuf:
0000000001000_0000000000000
[                          ] <- next range to compare
```

Because it is not easy to compare with non-specific part of a number, we copy the higher part of the number to its lower part after each comparison, if needed. Then we may always do the comparison on the lowest part of `tmpbuf`.
After cycle 2, we do assignment `tmpbuf[12:0] <= tmpbuf[25:13]` since `(tmpbuf[25:0] >= (1 << 13))` and results in:

```
msb_at_block:
11x
tmpbuf:
0000000001000_0000000001000
              [           ] <- next range to compare
```

At cycle 3, no copy operation involved since `(tmpbuf[12:0] < (1 << 7))`

```
msb_at_block:
110
tmpbuf:
0000000001000_0000000001000
                    [     ] <- next range to compare
```

At cycle 4, we search the range bit-by bit to find the correct position of the highest `1` in the original `B[51:0]` and store the result to `idxMsb`.

Clock cycle 5

Overhere, we have to explain why we computed `idxMsb` at **clock cycle 1-4**. When we want to generate the output for our `A`×`B` product, we have to find the highest `1` in `mprod` and hide it if the product is a normal number. In the case that `A` and `B` are both normal number, it is very easy because

$$1.\underbrace{xx\ldots xx}_{52} \quad \times \quad 1.\underbrace{xx\ldots xx}_{52}$$

always leads to one of the three:

$$01.\underbrace{xx\ldots xx}_{104} \tag{1}$$

$$10.\underbrace{xx\ldots xx}_{104} \tag{2}$$

$$11.\underbrace{xx\ldots xx}_{104} \tag{3}$$

However, on the other hand in the subnormal case, we also need to hide the highest `1` (if the product is normal) but we do not know where it is. Thus we were finding the highest `1` in `B[51:0]`, and this is nearly equivalent to search in `mprod`, which was not available at that time.

In this cycle, we only do the alignment for subnormal case:

```
if (subnormal)
    mprod <= mprod << idxMsb;
```

If `(!subnormal)`, do nothing.

Clock cycle 6

After clock 5, no matter the input contained subnormal number or not. The `mprod` is guaranteed to have the form in Equation (1), (2), or (3).
Then we need to check `mprod[105]`. If it is `1`, this means the multiplication produced a carry-out. We do 1-bit right-shift on `mprod` to ensure `(mprod[104] == 1)`.

Another big thing for this cycle is the computation of `expn`.

```
if (subnormal)
    expn <= sign_Aexpn - 11'd1022 - sign_idxMsb + sign_carry;
else
    expn <= sign_Aexpn + sign_Bexpn - 11'd1023 + sign_carry;
```

where the variables with "sign" prefix are just the variables with "signed" declaration to execute signed arithmetic operations.

Clock cycle 7

According to the value of `expn`, the output have four possibly formats:

- `expn` $\geq$ `11'b111_1111_1111`
  Since it exceeds the maximum range of valid `expn`, it should be rouned to ∞ according to the standard.
- `11'b111_1111_1111` > `expn` > 0
  Result in a normal number.
- 0 $\geq$ `expn` $\geq$ -52
  Results in a subnormal number.
- `-52` > `expn`
  Since the value is too tiny, it should be rounded to 0 according to the standard.

In this cycle, `mprod` is right-shifted to prepare for the subnormal output.

```
if (sign_zero >= expn && expn >= -52)
    mprod <= mprod >> (2 + ~expn);
```

where (2+ $\sim$ expn) is equivalent to (1 + |expn|) in two's complement system.

This `+1` is due to shift the implicit leading `1` at `mprod[104]` to fractional part, since there is no the leading `1` in subnormal format.

Clock cycle 8

This cycle is for rounding. Apply rounding to nearest according to the requirement.

```
{mprod[105], mprod[103:52]} <= mprod[103:52] + mprod[51];
```

The carry is put at `mprod[105]` because we may avoid usage of a logic MUX for `sign_carry`, which is assigned by

```
wire signed [1:0] sign_carry = {1'b0, mprod[105]};
```

Clock cycle 9

Cycle 9 is the last cycle in calculation stage. We need to generate the result in a correct format.

```
if (expn >= sign_0x7FF)
    mprod[103:52] <= 0;
else if (expn < -52)
    mprod[103:52] <= 0;
```

```
if (expn >= sign_0x7FF)
    expn[10:0] <= {11{1'b1}};
else if (expn > sign_zero)
    expn[10:0] <= expn + sign_carry;
else if (expn >= -52)
    expn[10:0] <= {{9{1'b0}}, sign_carry};
else
    expn[10:0] <= 0;
```

Notice that after the rounding in cycle 8, the `expn` may be added by `1`. A number in subnormal format may become a number in normal format. Nevertheless, we do not have to deal with it, because the `1` carried to `mprod[104]` exactly becomes the implicit leading `1` in normal format.

2 Design Test

In fact, before I started to write the verilog program, I wrote a C program to reach the desired multiplication. This helps me to design the algorithm in a more abstract level, without considering some RTL detail.

With the C program, I can generate arbitrary number of test cases by the C function rand(). I generated 100000 random patterns to test my verilog implementation. However, since the possibilities of appearance of subnormal numbers are too low, I set the first 1000 pattern as subnormal numbers by clearing their exponential field to 0. In this way, the testbench can be made very simple.

1. Generate many input `A` and `B` by C program.
2. Compute the product by the C program.
3. Compute the product by my verilog design.
4. Compare the two products above.

In my testbench, the output of my verilog program are actually compared with both results from C and results from Verilog built-in floating multiplication.
The main drawback of my random-generated patterns is that the patterns are very often to become infinity since the exponential field of the input are arbitrary.

```
ncsim> source /usr/INCISIVE15/tools/inca/files/ncsimrc
ncsim> run
Total error:           0 /     100000
Simulation complete via $finish(1) at time 176190750 NS + 0
./TEST.v:99     $finish;
ncsim> exit
```

* Pattern Location: `/misc/Si2_RAID-1/COURSE/dic/dic03/final_project/Pattern`
* Testbench Location: `/misc/Si2_RAID-1/COURSE/dic/dic03/final_project/TEST.v`

3 Implementation Result

3.1 Post-Synthesis Results

pwd: /misc/Si2_RAID-1/COURSE/dic/dic03/final_project/Lowpower
netlist: /misc/Si2_RAID-1/COURSE/dic/dic03/final_project/Lowpower/fp_mult_synLP.v

- Area

Instance	Module	Cell Count	Cell Area	Net Area	Total Area
fp_mult		4475	127530.850	68126.621	195657.471

- Power
  - Total: 0.63 (mW)

Category	Leakage	Internal	Switching	Total	Row%
memory	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00	0.00%
register	1.68908e−06	1.57490e−04	3.45403e−05	1.93720e−04	30.60%
latch	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00	0.00%
logic	1.30044e−05	2.17370e−04	1.82472e−04	4.12846e−04	65.22%
bbox	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00	0.00%
clock	7.05780e−08	1.40089e−05	1.23809e−05	2.64604e−05	4.18%
pad	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00	0.00%
pm	0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00	0.00%
Subtotal	1.47641e−05	3.88869e−04	2.29393e−04	6.33026e−04	100.00%
Percentage	2.33%	61.43%	36.24%	100.00%	100.00%

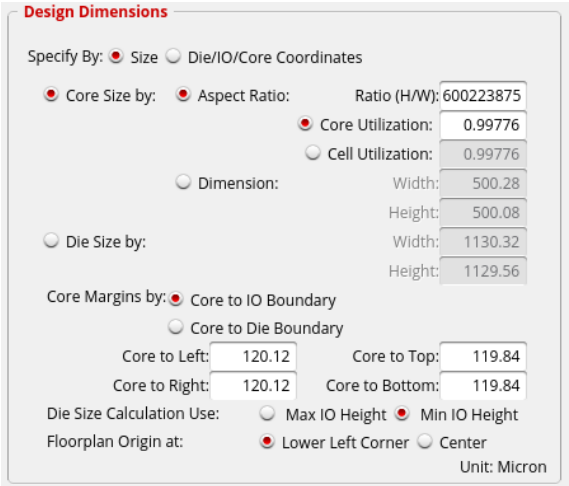
- Timing

```
Cost Group   : 'CLK' (path_group 'CLK')
Timing slack :    1376ps
Start-point  : calcount_reg[3]/CK
End-point    : mprod_reg[105]/SI
```

3.2 Post-Layout Results

pwd: /misc/Si2_RAID-1/COURSE/dic/dic03/final_project/APR
pwd: /misc/Si2_RAID-1/COURSE/dic/dic03/final_project/DRCLVS
CHIP.gds: /misc/Si2_RAID-1/COURSE/dic/dic03/final_project/APR/CHIP.gds
CHIP.v: /misc/Si2_RAID-1/COURSE/dic/dic03/final_project/APR/CHIP_postLayout.v
CHIP_LVS.v: /misc/Si2_RAID-1/COURSE/dic/dic03/final_project/APR/CHIP_LVS.v
CHIP.sdf: /misc/Si2_RAID-1/COURSE/dic/dic03/final_project/APR/CHIP.sdf

- Area
  - CHIP Width: 1130.32 (μm)
  - CHIP Height: 1129.56 (μm)



- Power
  - Total: 2.77 (mW)

Cell Name	Internal Power	Switching Power	Total Power	Leakage Power
Total (4489 of 4531)	2.194	0.5584	2.77	0.01718

- Timing

```
Analysis View: CHECK_SETUP_TIME
Other End Arrival Time          1.481
- Setup                         0.853
+ Phase Shift                  50.000
+ CPPR Adjustment               0.000
= Required Time                50.628
- Arrival Time                 48.681
= Slack Time                    1.947

Analysis View: CHECK_HOLD_TIME
Other End Arrival Time         25.506
+ Clock Gating Hold             0.000
+ Phase Shift                   0.000
- CPPR Adjustment               0.639
= Required Time                24.867
  Arrival Time                 25.013
  Slack Time                    0.146
```

- Simulation Result
- LVS Result
- Layout

```
cad2 [final_project/APR]% ncverilog -f sim.f
ncverilog(64): 15.20-s086: (c) Copyright 1995-2020 Cadence Design Systems, Inc.
Loading snapshot worklib.TEST:v .................... Done
*Verdi* Loading libsscore_ius152.so
*** Registering RTL Compiler Low Power PLI
ncsim> source /usr/INCISIVE15/tools/inca/files/ncsimrc
ncsim> run
Total error:           0 /     100000
Simulation complete via $finish(1) at time 176190750 NS + 0
./TEST.v:101     $finish;
ncsim> exit
```

```
                     OVERALL COMPARISON RESULTS


            #        ###################       _   _
           #         #                 #       *   *
      #   #          #     CORRECT     #         |
       # #           #                 #       \___/
        #            ###################

Warning:  Ambiguity points were found and resolved arbitrarily.
Warning:  LVS property resolution maximum exceeded.
```